Expressions

Essentials of Programming Languages (Chapter 3)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Expressions

- In this chapter, we study the binding and scoping of variables
- We write specifications for our languages, and implement them using interpreters, following the interpreter recipe
- Our specifications and interpreters take a context argument, called the *environment*, which keeps track of the meaning of each variable in the expression being evaluated

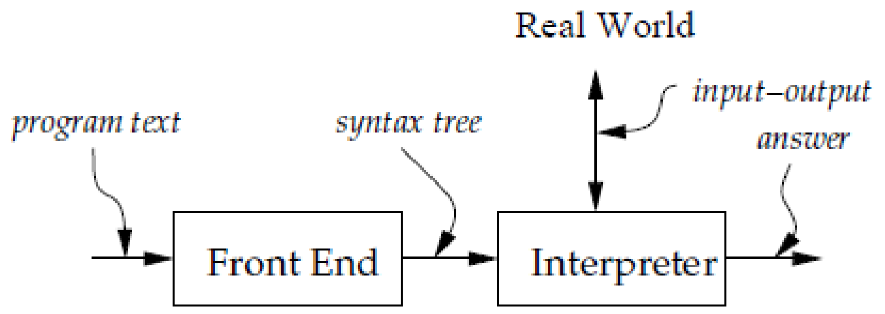
Specification and Implementation Strategy

Our specification will consist of assertions of the form

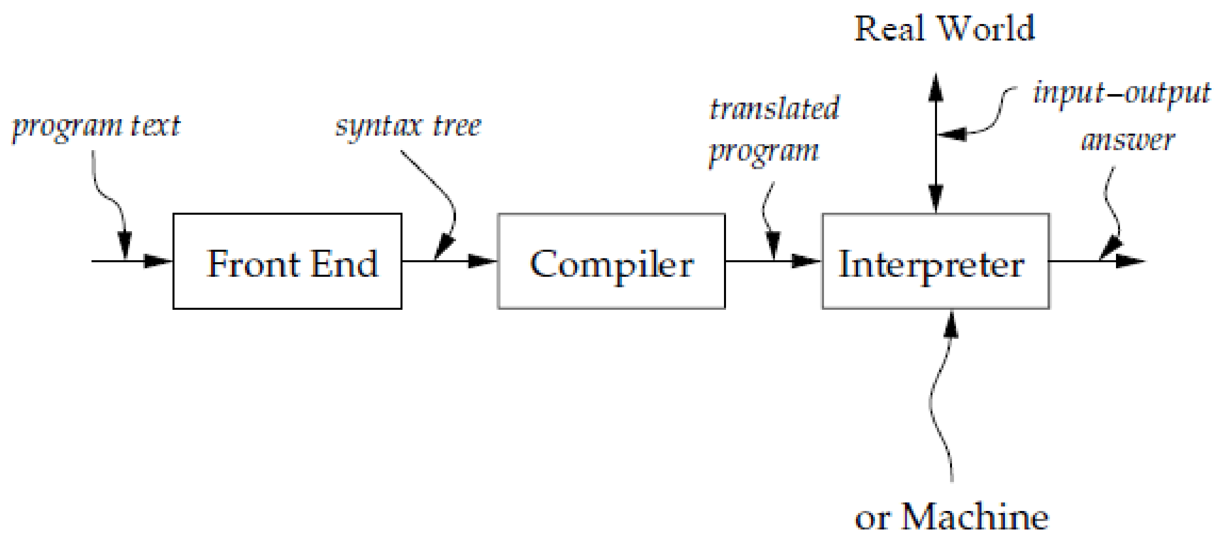
$$(\texttt{value-of}\ \ exp\ \ \rho) = val$$

meaning that the value of expression *exp* in environment ρ should be *val*. We write down rules of inference and equations, like those in chapter 1, that will enable us to derive such assertions. We use the rules and equations by hand to find the intended value of some expressions.

Figure 3.1 Block diagrams for a language-processing system



(a) Execution via interpreter



Language Processing System

- We start with the text of the program written in the language we are implementing
  - This is called the source language or the defined language
- Program text (a program in the source language) is passed through a front end
  - It converts the program to an abstract syntax tree
- The interpreter is itself written in some language
  - We call that language the implementation language or the defining language
- In figure 3.1(b), the interpreter is replaced by a compiler, which translates the abstract syntax tree into a program in some other language (the target language), and that program is executed

LET: A Simple Language

Program ::= *Expression*
$\boxed{\texttt{a-program (exp1)}}$

Expression ::= *Number*
$\boxed{\texttt{const-exp (num)}}$

Expression ::= `-` `(`*Expression* `,` *Expression*`)`
$\boxed{\texttt{diff-exp (exp1 exp2)}}$

Expression ::= `zero?` `(`*Expression*`)`
$\boxed{\texttt{zero?-exp (exp1)}}$

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
$\boxed{\texttt{if-exp (exp1 exp2 exp3)}}$

Expression ::= *Identifier*
$\boxed{\texttt{var-exp (var)}}$

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
$\boxed{\texttt{let-exp (var exp1 body)}}$

```
(scan&parse "-(55, -(x,11))")
#(struct:a-program
   #(struct:diff-exp
      #(struct:const-exp 55)
      #(struct:diff-exp
         #(struct:var-exp x)
         #(struct:const-exp 11))))
```

Specification of Values

- An important part of the specification of any programming language is the set of values that the language manipulates
- Each language has at least two such sets:
  - The expressed values
    - They are the possible values of expressions
  - The denoted values
    - They are the values bound to variables

$$\begin{aligned} ExpVal &= Int + Bool \\ DenVal &= Int + Bool \end{aligned}$$

num-val	$: Int \rightarrow ExpVal$
bool-val	$: Bool \rightarrow ExpVal$
expval->num	$: ExpVal \rightarrow Int$
expval->bool	$: ExpVal \rightarrow Bool$

Environments

An environment is a function whose domain is a finite set of variables and whose range is the denoted values. We use some abbreviations when writing about environments.

- ρ ranges over environments.
- $[]$ denotes the empty environment.
- $[var = val]\rho$ denotes `(extend-env` var val ρ`)`.
- $[var_1 = val_1, var_2 = val_2]\rho$ abbreviates $[var_1 = val_1]([var_2 = val_2]\rho)$, etc.
- $[var_1 = val_1, var_2 = val_2, \ldots]$ denotes the environment in which the value of var_1 is val_1, etc.

Environments

We will occasionally write down complicated environments using indentation to improve readability. For example, we might write

```
[x=3]
 [y=7]
  [u=5] ρ
```

to abbreviate

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 ρ)))
```

Specifying the Behavior of Expressions

```
(value-of (const-exp n) ρ) = (num-val n)

(value-of (var-exp var) ρ) = (apply-env ρ var)

(value-of (diff-exp exp₁ exp₂) ρ)
= (num-val
     (-
       (expval->num (value-of exp₁ ρ))
       (expval->num (value-of exp₂ ρ))))
```

constructors:

const-exp	: $Int \to Exp$
zero?-exp	: $Exp \to Exp$
if-exp	: $Exp \times Exp \times Exp \to Exp$
diff-exp	: $Exp \times Exp \to Exp$
var-exp	: $Var \to Exp$
let-exp	: $Var \times Exp \times Exp \to Exp$

observer:

value-of	: $Exp \times Env \to ExpVal$

…

We write «exp» to denote the AST for expression exp. We also write $\lceil n \rceil$ in place of `(num-val` n`)`, and $\lfloor val \rfloor$ in place of `(expval->num` val`)`. We will also use the fact that $\lfloor \lceil n \rceil \rfloor = n$.

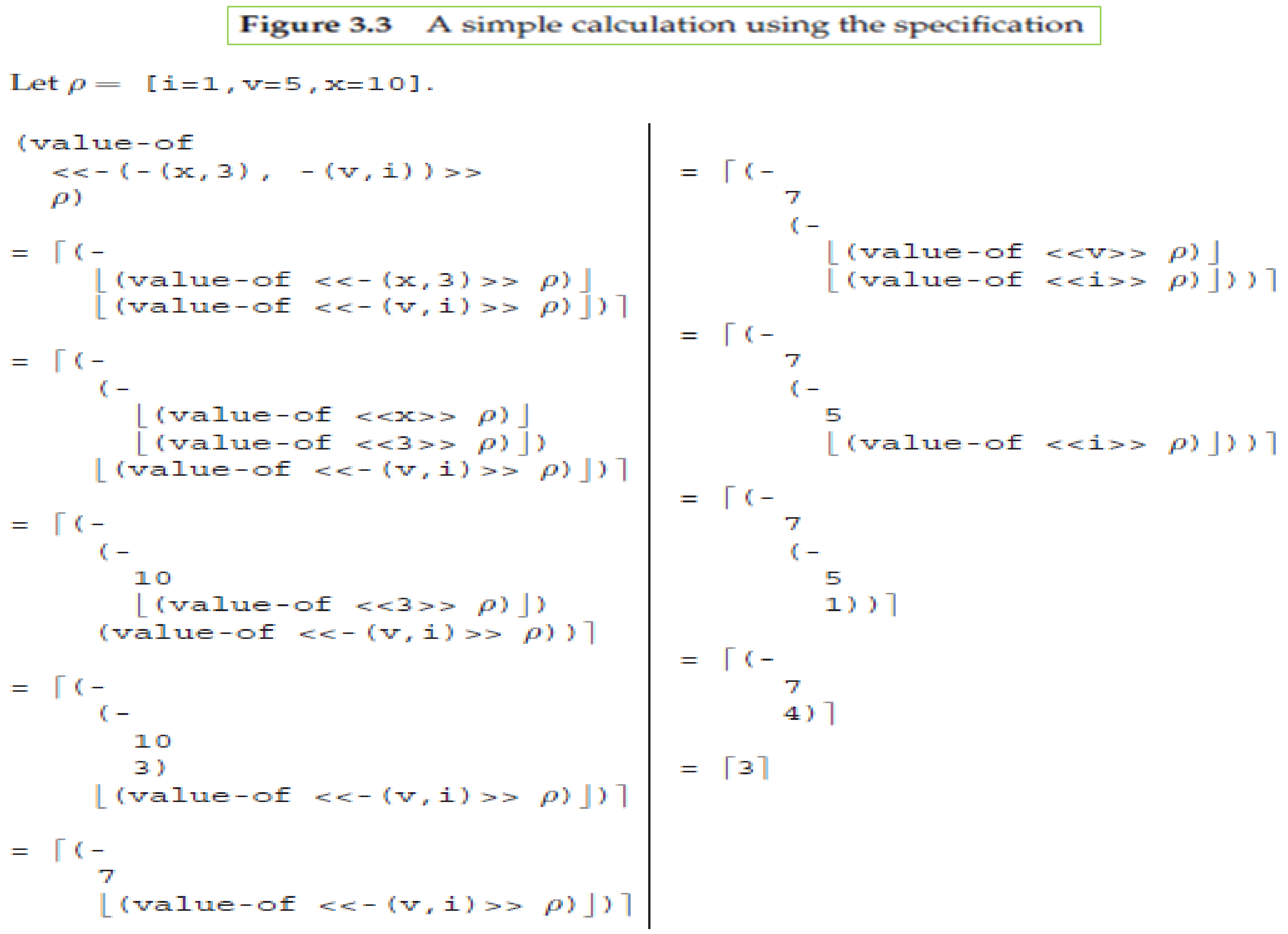


Figure 3.3 A simple calculation using the specification

Let $\rho =$ `[i=1,v=5,x=10]`.

```
(value-of
  <<-(-(x,3), -(v,i))>>
  ρ)

= ⌈(-
    ⌊(value-of <<-(x,3)>> ρ)⌋
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      ⌊(value-of <<x>> ρ)⌋
      ⌊(value-of <<3>> ρ)⌋)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      10
      ⌊(value-of <<3>> ρ)⌋)
    (value-of <<-(v,i)>> ρ))⌉

= ⌈(-
    (-
      10
      3)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    7
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉
```

```
= ⌈(-
      7
      (-
        ⌊(value-of <<v>> ρ)⌋
        ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
      7
      (-
        5
        ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
      7
      (-
        5
        1))⌉

= ⌈(-
      7
      4)⌉

= ⌈3⌉
```

Specifying the Behavior of Programs

In our language, a whole program is just an expression. In order to find the value of such an expression, we need to specify the values of the free variables in the program. So the value of a program is just the value of that expression in a suitable initial environment. We choose our initial environment to be `[i=1,v=5,x=10]`.

```
  (value-of-program exp)
= (value-of exp [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉])
```

Specifying Conditionals

We use `bool-val` as a constructor to turn a boolean into an expressed value, and `expval->num` as an extractor to check whether an expressed value is an integer, and if so, to return the integer.

$$\frac{(\texttt{value-of}\ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{zero?-exp}\ exp_1)\ \ \rho) \\ \quad = \begin{cases}(\texttt{bool-val \#t}) & \text{if } (\texttt{expval->num}\ val_1) = 0 \\ (\texttt{bool-val \#f}) & \text{if } (\texttt{expval->num}\ val_1) \neq 0\end{cases}\end{array}}$$

$$\frac{(\texttt{value-of}\ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \ \rho) \\ \quad = \begin{cases}(\texttt{value-of}\ exp_2\ \ \rho) & \text{if } (\texttt{expval->bool}\ val_1) = \texttt{\#t} \\ (\texttt{value-of}\ exp_3\ \ \rho) & \text{if } (\texttt{expval->bool}\ val_1) = \texttt{\#f}\end{cases}\end{array}}$$

…

Rules of inference like this make the intended behavior of any individual expression easy to specify, but they are not very good for displaying a deduction.

We can then use substitution of equals for equals to display a calculation.

For an `if-exp`, the equational specification is

```
(value-of (if-exp exp₁ exp₂ exp₃) ρ)
= (if (expval->bool (value-of exp₁ ρ))
      (value-of exp₂ ρ)
      (value-of exp₃ ρ))
```

Figure 3.4 A simple calculation for a conditional expression

Let $\rho =$ `[x=⌈33⌉,y=⌈22⌉]`.

```
(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)

= ⌈18⌉
```

Specifying let

let expressions may be nested, as in

```
let z = 5
in let x = 3
   in let y = -(x,1)      % here x = 3
      in let x = 4
         in -(z, -(x,y)) % here x = 4
```

In this example, the reference to x in the first difference expression refers to the outer declaration, whereas the reference to x in the other difference expression refers to the inner declaration, and thus the entire expression's value is 3.

…

The right-hand side of the `let` is also an expression, so it can be arbitrarily complex. For example,

```
let x = 7
in let y = 2
   in let y = let x = -(x,1)
              in -(x,y)
      in -(-(x,8), y)
```

Here the x declared on the third line is bound to 6, so the value of y is 4, and the value of the entire expression is $((-1) - 4) = -5$.

…

We can write down the specification as a rule.

$$\frac{(\texttt{value-of}\ \mathit{exp}_1\ \rho) = \mathit{val}_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{let-exp}\ \mathit{var}\ \mathit{exp}_1\ \mathit{body})\ \rho)\\ \quad = (\texttt{value-of}\ \mathit{body}\ [\mathit{var} = \mathit{val}_1]\rho)\end{array}}$$

As before, it is often more convenient to recast this as the equation

$$\begin{array}{l}(\texttt{value-of}\ (\texttt{let-exp}\ \mathit{var}\ \mathit{exp}_1\ \mathit{body})\ \rho)\\ = (\texttt{value-of}\ \mathit{body}\ [\mathit{var}=(\texttt{value-of}\ \mathit{exp}_1\ \rho)]\rho)\end{array}$$

Figure 3.5 An example of `let`

```
(value-of
  <<let x = 7
    in let y = 2
       in let y = let x = -(x,1) in -(x,y)
          in -(-(x,8),y)>>
  ρ0)

= (value-of
    <<let y = 2
      in let y = let x = -(x,1) in -(x,y)
         in -(-(x,8),y)>>
    [x=⌈7⌉]ρ0)

= (value-of
    <<let y = let x = -(x,1) in -(x,y)
      in -(-(x,8),y)>>
    [y=⌈2⌉][x=⌈7⌉]ρ0)
```

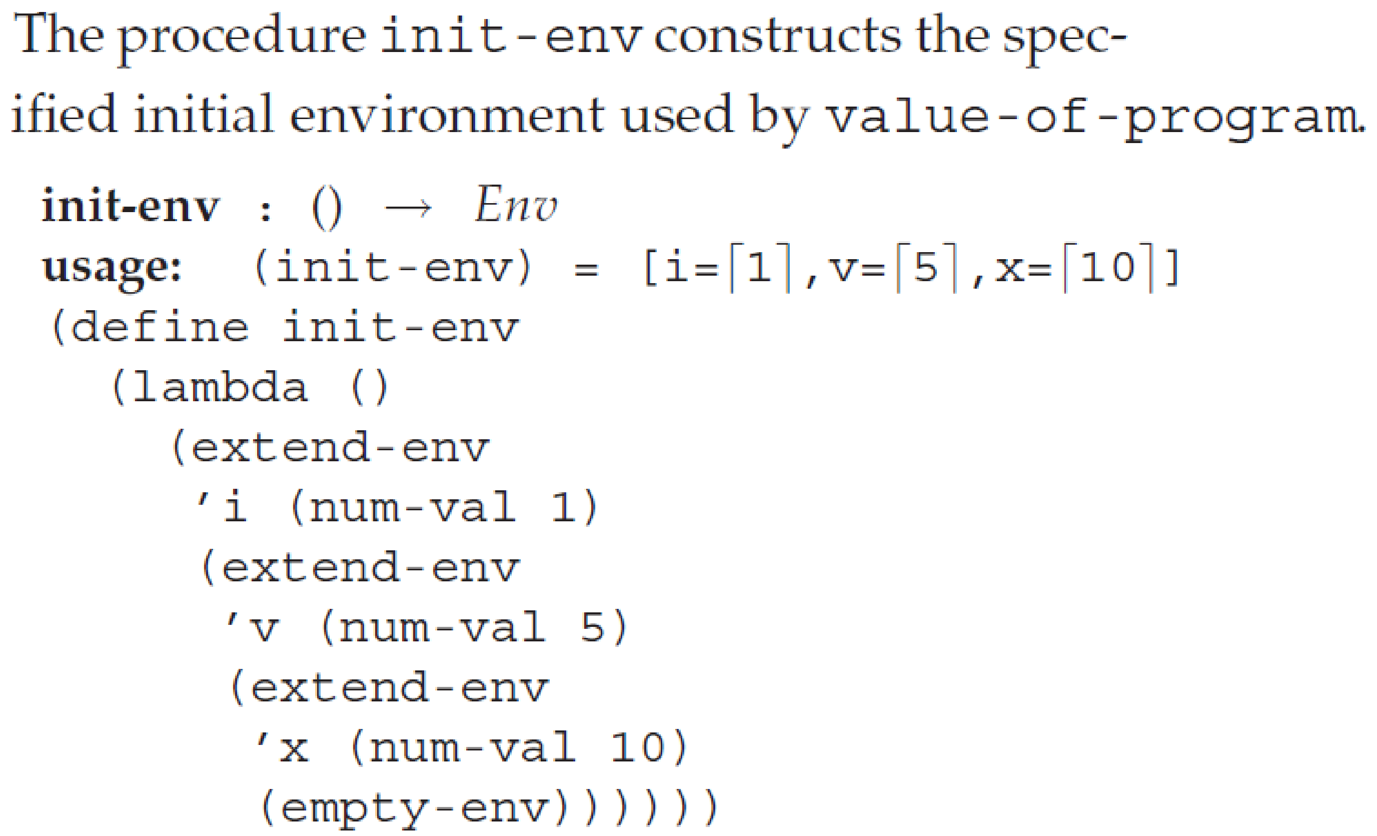
Let ρ_1 = `[y=`$\lceil 2 \rceil$`]` `[x=`$\lceil 7 \rceil$`]` ρ_0.

```
= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<let x = -(x,1) in -(x,y)>> ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=(value-of <<-(x,1)>> ρ1)]ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=⌈6⌉]ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=⌈4⌉]ρ1)

= ⌈(- (- 7 8) 4)⌉

= ⌈-5⌉
```

Figure 3.6 Syntax data types for the LET language

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?)))
```

Implementing the Specification of LET



The procedure `init-env` constructs the specified initial environment used by `value-of-program`.

init-env : () $\rightarrow$ *Env*

usage: `(init-env) = [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉]`

```
(define init-env
  (lambda ()
    (extend-env
     'i (num-val 1)
     (extend-env
      'v (num-val 5)
      (extend-env
       'x (num-val 10)
       (empty-env))))))
```

Figure 3.7 Expressed values for the LET language

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?)))
```

expval->num : $ExpVal \to Int$

```
(define expval->num
  (lambda (val)
    (cases expval val
      (num-val (num) num)
      (else (report-expval-extractor-error 'num val)))))
```

expval->bool : $ExpVal \to Bool$

```
(define expval->bool
  (lambda (val)
    (cases expval val
      (bool-val (bool) bool)
      (else (report-expval-extractor-error 'bool val)))))
```

Figure 3.8 Interpreter for the LET language

run : *String* → *ExpVal*

```
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))
```

value-of-program : *Program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

value-of : *Exp* × *Env* → *ExpVal*

```
(define value-of
  (lambda (exp env)
    (cases expression exp
```

$$\boxed{\texttt{(value-of (const-exp } n\texttt{) } \rho\texttt{)} = n}$$

```
      (const-exp (num) (num-val num))
```

$$\boxed{\texttt{(value-of (var-exp } var\texttt{) } \rho\texttt{)} = \texttt{(apply-env } \rho\ var\texttt{)}}$$

```
      (var-exp (var) (apply-env env var))
```

$$\boxed{\begin{array}{l}\texttt{(value-of (diff-exp } exp_1 \ exp_2\texttt{)} \ \rho\texttt{)} = \\ \quad \lceil \texttt{(- } \lfloor \texttt{(value-of } exp_1 \ \rho\texttt{)} \rfloor \ \lfloor \texttt{(value-of } exp_2 \ \rho\texttt{)} \rfloor \texttt{)} \rceil \end{array}}$$

```
(diff-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((num1 (expval->num val1))
          (num2 (expval->num val2)))
      (num-val
        (- num1 num2)))))
```

$$\boxed{\begin{array}{c}\texttt{(value-of } exp_1 \ \rho) = val_1 \\ \hline \begin{array}{l}\texttt{(value-of (zero?-exp } exp_1\texttt{)} \ \rho\texttt{)} \\ \quad = \begin{cases}\texttt{(bool-val \#t)} & \text{if } \texttt{(expval->num } val_1\texttt{)} = 0 \\ \texttt{(bool-val \#f)} & \text{if } \texttt{(expval->num } val_1\texttt{)} \neq 0\end{cases}\end{array}\end{array}}$$

```
(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (if (zero? num1)
        (bool-val #t)
        (bool-val #f)))))
```

$$\frac{(\texttt{value-of}\ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \rho) \\ \quad = \begin{cases}(\texttt{value-of}\ exp_2\ \rho) & \text{if } (\texttt{expval->bool}\ val_1) = \texttt{\#t} \\ (\texttt{value-of}\ exp_3\ \rho) & \text{if } (\texttt{expval->bool}\ val_1) = \texttt{\#f}\end{cases}\end{array}}$$

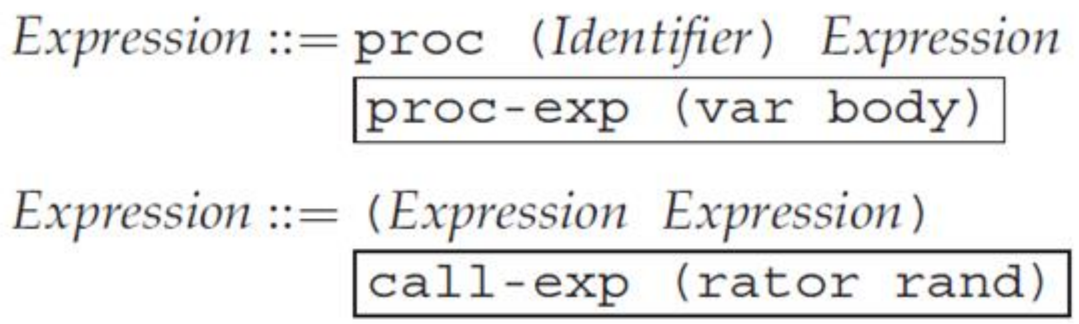
```
(if-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env)))
    (if (expval->bool val1)
      (value-of exp2 env)
      (value-of exp3 env))))
```

$$\frac{(\texttt{value-of}\ exp_1\ \ \rho) = val_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{let-exp}\ var\ exp_1\ body)\ \rho) \\ \quad = (\texttt{value-of}\ body\ [var = val_1]\rho)\end{array}}$$

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))))
```

PROC: A Language with Procedures

We need syntax for procedure creation and calling. This is given by the productions:

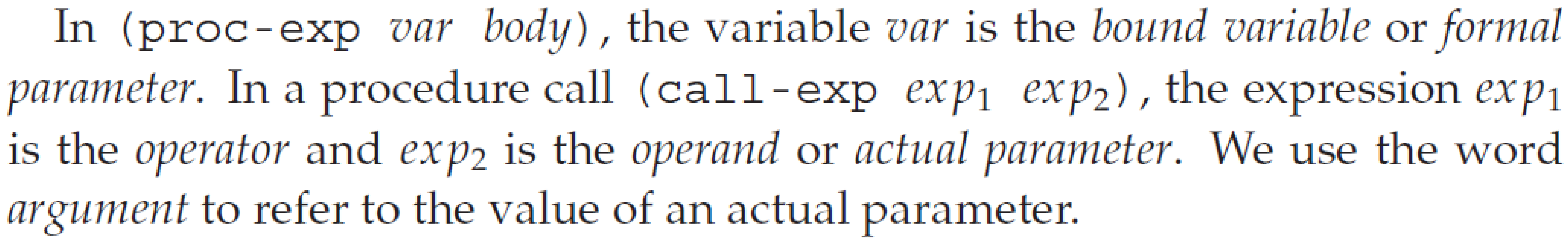


Expression ::= `proc` (*Identifier*) *Expression*

$\boxed{\texttt{proc-exp (var body)}}$

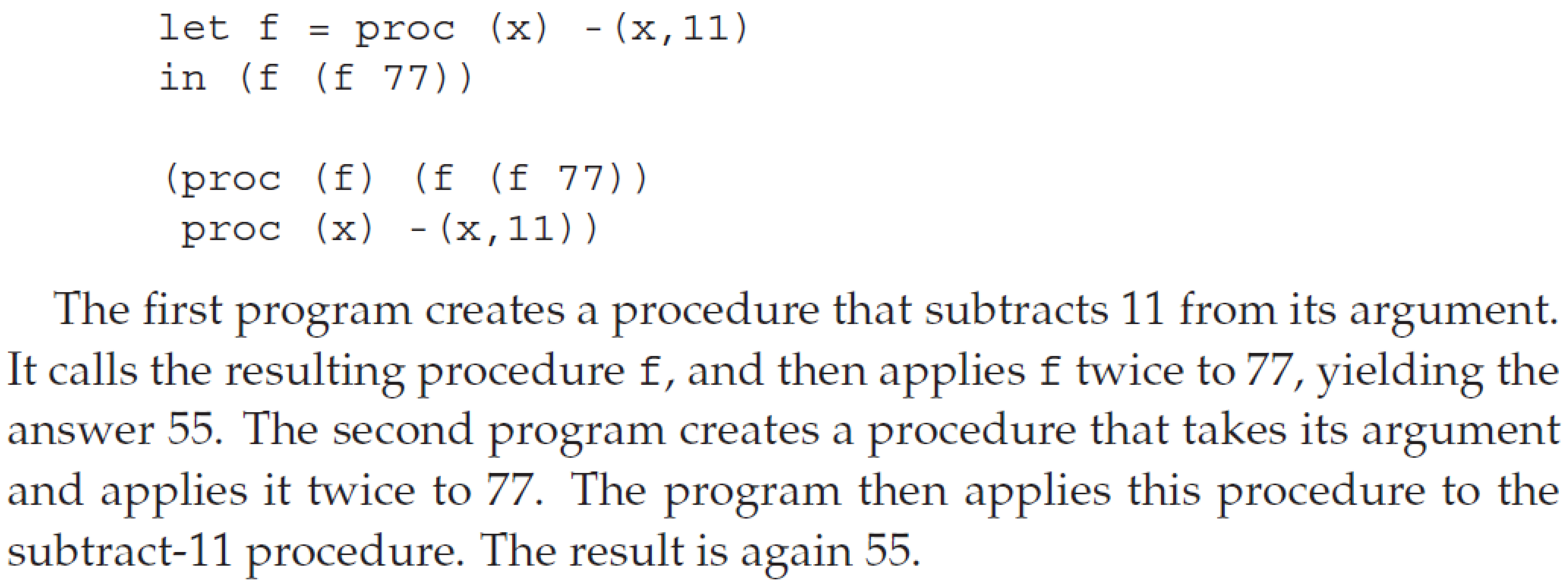
Expression ::= (*Expression* *Expression*)

$\boxed{\texttt{call-exp (rator rand)}}$



In (`proc-exp` *var* *body*), the variable *var* is the *bound variable* or *formal parameter*. In a procedure call (`call-exp` exp_1 exp_2), the expression exp_1 is the *operator* and exp_2 is the *operand* or *actual parameter*. We use the word *argument* to refer to the value of an actual parameter.

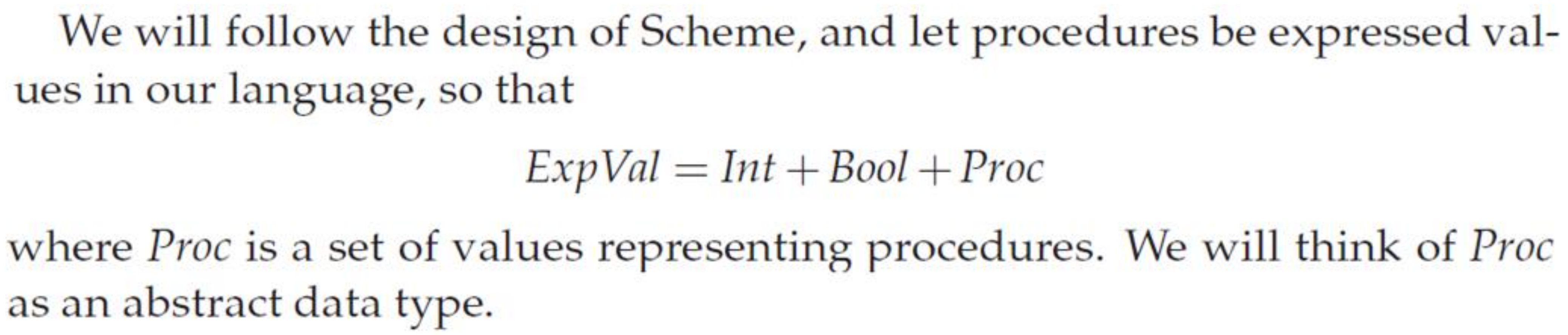
PROC: Example



```
let f = proc (x) -(x,11)
in (f (f 77))

(proc (f) (f (f 77))
 proc (x) -(x,11))
```

The first program creates a procedure that subtracts 11 from its argument. It calls the resulting procedure `f`, and then applies `f` twice to 77, yielding the answer 55. The second program creates a procedure that takes its argument and applies it twice to 77. The program then applies this procedure to the subtract-11 procedure. The result is again 55.

PROC



We will follow the design of Scheme, and let procedures be expressed values in our language, so that

$$ExpVal = Int + Bool + Proc$$

where *Proc* is a set of values representing procedures. We will think of *Proc* as an abstract data type.

PROC

Expression ::= `proc` (*Identifier*) *Expression*

$\boxed{\texttt{proc-exp (var body)}}$

Expression ::= (*Expression* *Expression*)

$\boxed{\texttt{call-exp (rator rand)}}$

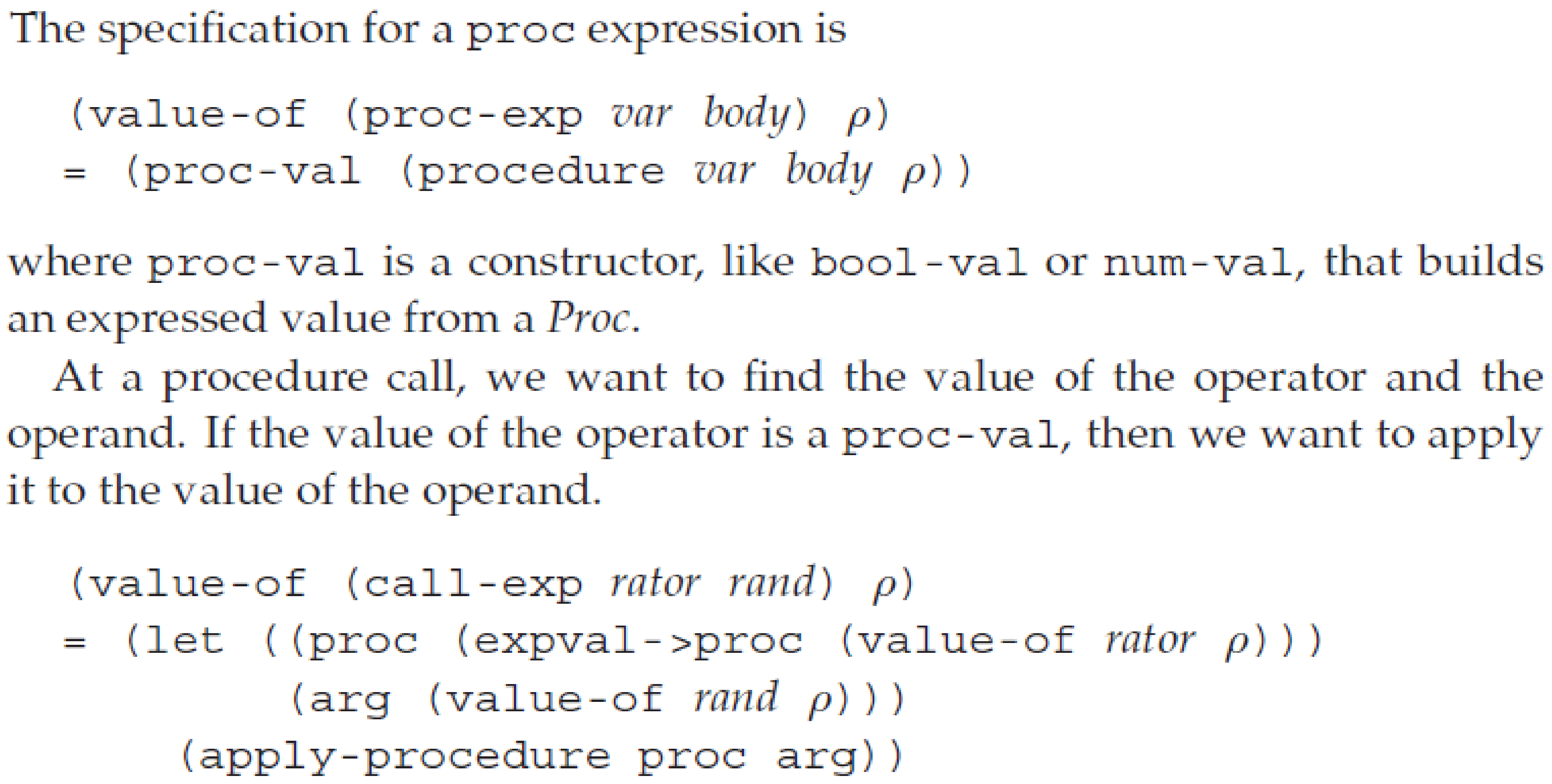
The lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call. Variables occurring free in the procedure should also obey the lexical binding rule.

PROC: Example

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

Here we evaluate the expression `proc (z) -(z,x)` twice. The first time we do it, `x` is bound to 200, so by the lexical scope rule, the result is a procedure that subtracts 200 from its argument. We name this procedure `f`. The second time we do it, `x` is bound to 100, so the resulting procedure should subtract 100 from its argument. We name this procedure `g`.

PROC: Specification



The specification for a `proc` expression is

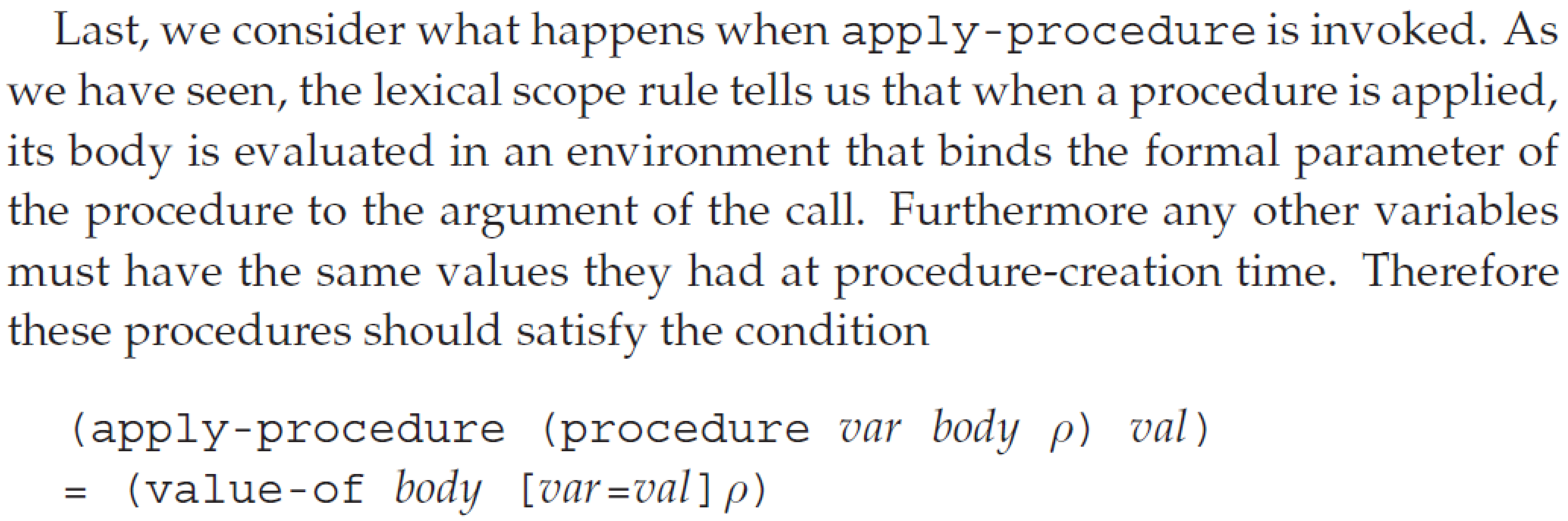
```
(value-of (proc-exp var body) ρ)
= (proc-val (procedure var body ρ))
```

where `proc-val` is a constructor, like `bool-val` or `num-val`, that builds an expressed value from a *Proc*.

At a procedure call, we want to find the value of the operator and the operand. If the value of the operator is a `proc-val`, then we want to apply it to the value of the operand.

```
(value-of (call-exp rator rand) ρ)
= (let ((proc (expval->proc (value-of rator ρ)))
        (arg (value-of rand ρ)))
    (apply-procedure proc arg))
```

PROC: Specification



Last, we consider what happens when `apply-procedure` is invoked. As we have seen, the lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call. Furthermore any other variables must have the same values they had at procedure-creation time. Therefore these procedures should satisfy the condition

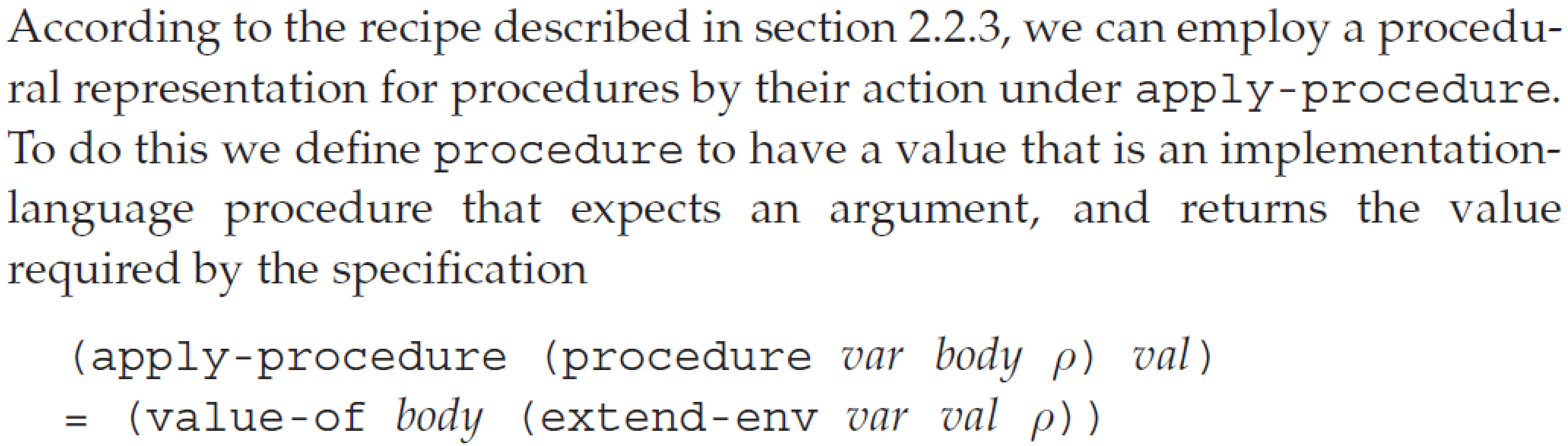
```
  (apply-procedure (procedure var body ρ) val)
= (value-of body [var=val]ρ)
```

```
(value-of
 <<let x = 200
   in let f = proc (z) -(z,x)
      in let x = 100
         in let g = proc (z) -(z,x)
            in -((f 1), (g 1))>>
 ρ)

= (value-of
   <<let f = proc (z) -(z,x)
     in let x = 100
        in let g = proc (z) -(z,x)
           in -((f 1), (g 1))>>
    [x=⌈200⌉]ρ)


= (value-of
   <<let x = 100
     in let g = proc (z) -(z,x)
        in -((f 1), (g 1))>>
    [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
     [x=⌈200⌉]ρ)

= (value-of
   <<let g = proc (z) -(z,x)
     in -((f 1), (g 1))>>
    [x=⌈100⌉]
     [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
      [x=⌈200⌉]ρ)
```

```
= (value-of
    <<-((f 1), (g 1))>>
    [g=(proc-val (procedure z <<-(z,x)>>
                   [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ))]
     [x=⌈100⌉]
      [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
       [x=⌈200⌉]ρ)

= ⌈(-
    (value-of <<(f 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ)
    (value-of <<(g 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ))⌉

= ⌈(-
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈200⌉]ρ)
      ⌈1⌉)
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈100⌉] [f=...] [x=⌈200⌉]ρ)
      ⌈1⌉))⌉
```

```
= ⌈(-
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈200⌉]ρ)
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈100⌉][f=...][x=⌈200⌉]ρ))⌉

= ⌈(- -199 -99)⌉

= ⌈-100⌉
```

PROC: Procedural Representation



According to the recipe described in section 2.2.3, we can employ a procedural representation for procedures by their action under `apply-procedure`. To do this we define `procedure` to have a value that is an implementation-language procedure that expects an argument, and returns the value required by the specification

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))
```

PROC: Procedural Representation

Therefore the entire implementation is

procedure : $Var \times Exp \times Env \rightarrow Proc$

```
(define procedure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))
```

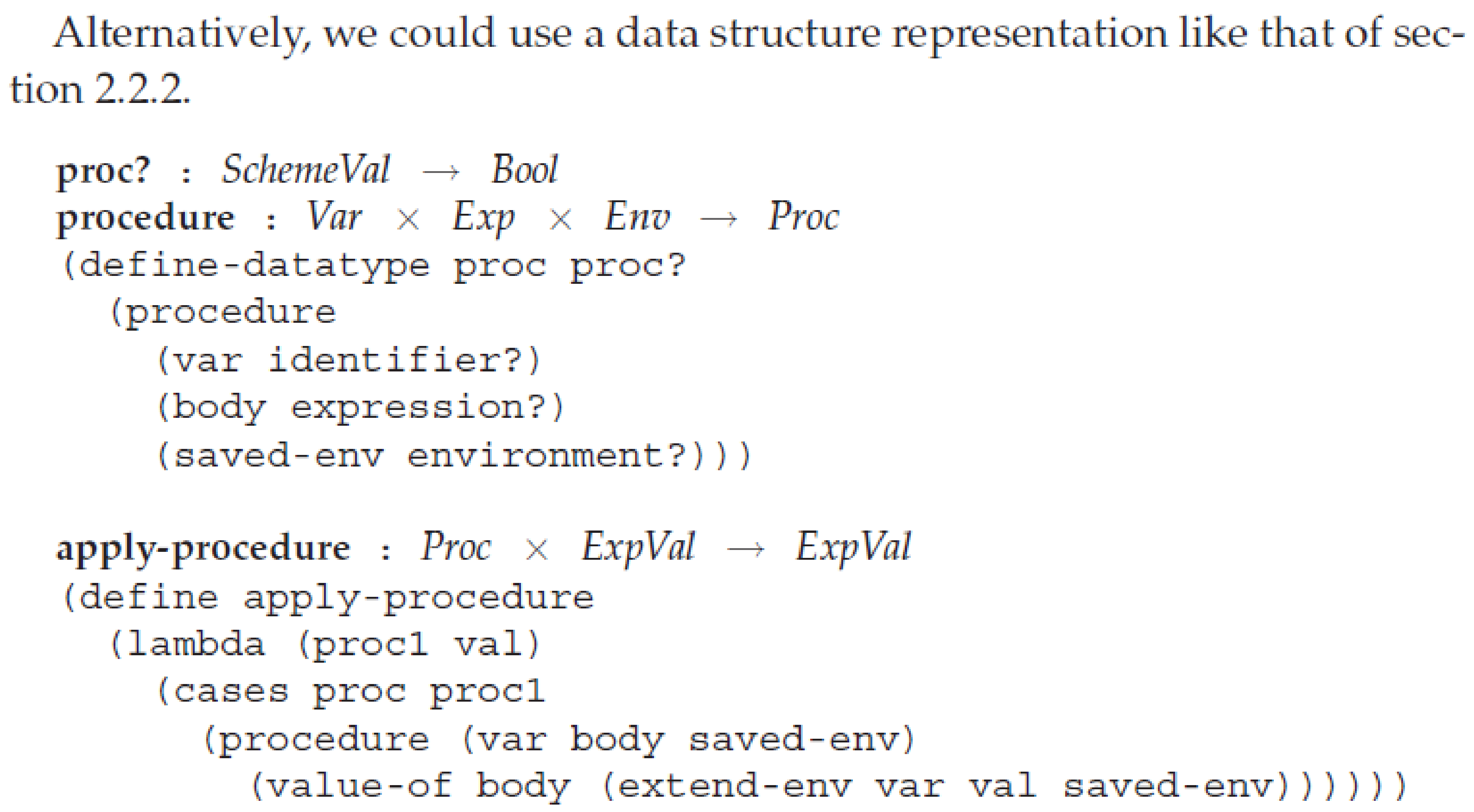
apply-procedure : $Proc \times ExpVal \rightarrow ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (proc1 val)))
```

proc? : $SchemeVal \rightarrow Bool$

```
(define proc?
  (lambda (val)
    (procedure? val)))
```

PROC: define-datatype



Alternatively, we could use a data structure representation like that of section 2.2.2.

$\textbf{proc?} : \mathit{SchemeVal} \rightarrow \mathit{Bool}$
$\textbf{procedure} : \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \rightarrow \mathit{Proc}$

```
(define-datatype proc proc?
  (procedure
    (var identifier?)
    (body expression?)
    (saved-env environment?)))
```

$\textbf{apply-procedure} : \mathit{Proc} \times \mathit{ExpVal} \rightarrow \mathit{ExpVal}$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

…

In either implementation, we add an alternative to the data type expval

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
  (proc-val
    (proc proc?)))
```

and we need to add two new clauses to value-of

```
(proc-exp (var body)
  (proc-val (procedure var body env)))

(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc arg)))
```

LETREC: A Language with Recursive Procedures

For example:

```
letrec double(x)
        = if zero?(x) then 0 else -((double -(x,1)), -2)
in (double 6)
```

The left-hand side of a recursive declaration is the name of the recursive procedure and its bound variable. To the right of the = is the procedure body. The production for this is

Expression ::= `letrec` *Identifier* `(`*Identifier*`)` `=` *Expression* `in` *Expression*

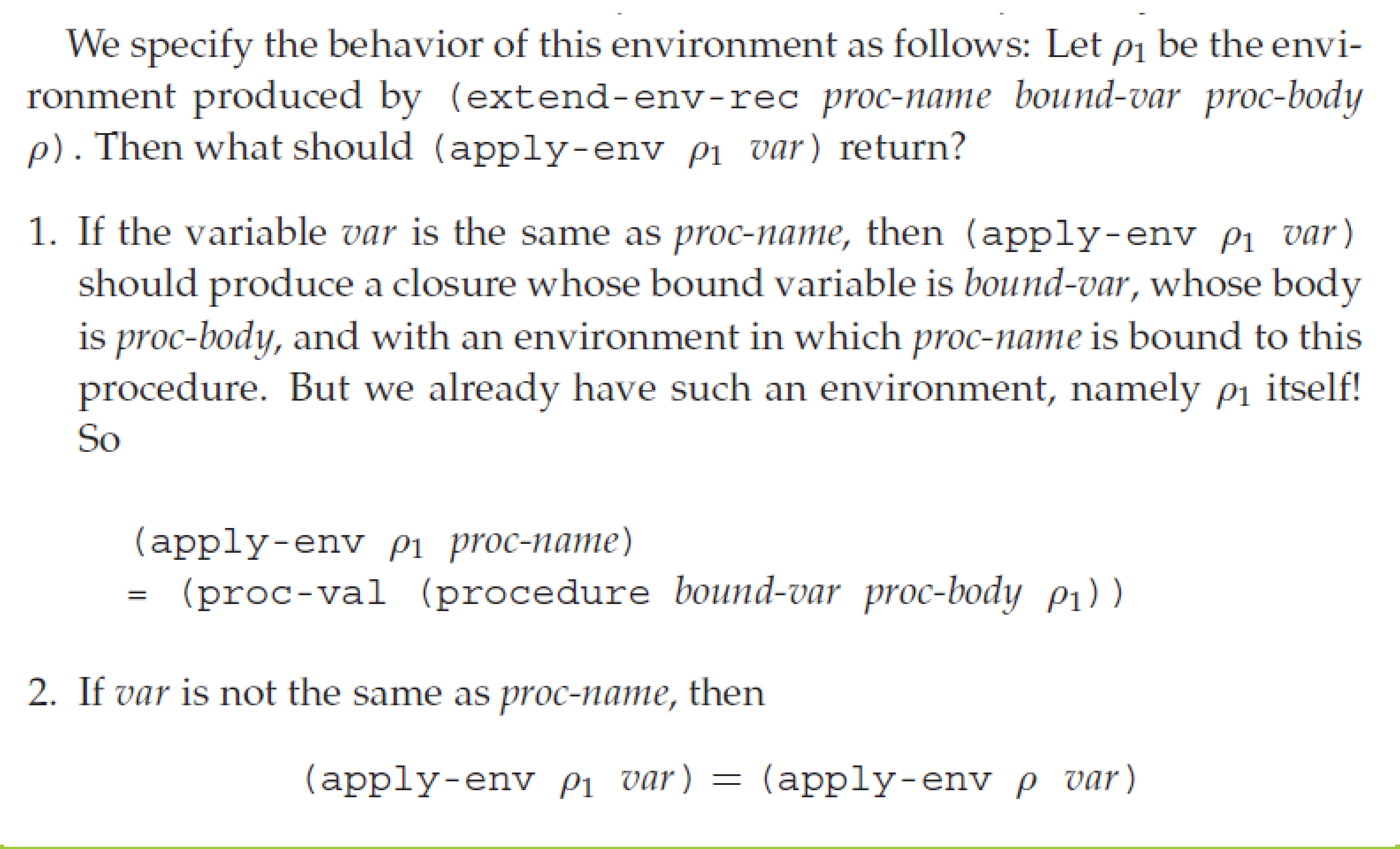
`letrec-exp (p-name b-var p-body letrec-body)`

LETREC: Specification

The value of a `letrec` expression is the value of the body in an environment that has the desired behavior:

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

LETREC: Specification



We specify the behavior of this environment as follows: Let ρ_1 be the environment produced by `(extend-env-rec` *proc-name bound-var proc-body* ρ`)`. Then what should `(apply-env` ρ_1 *var*`)` return?

1. If the variable *var* is the same as *proc-name*, then `(apply-env` ρ_1 *var*`)` should produce a closure whose bound variable is *bound-var*, whose body is *proc-body*, and with an environment in which *proc-name* is bound to this procedure. But we already have such an environment, namely ρ_1 itself! So

```
(apply-env $\rho_1$ proc-name)
= (proc-val (procedure bound-var proc-body $\rho_1$))
```

2. If *var* is not the same as *proc-name*, then

```
(apply-env $\rho_1$ var) = (apply-env $\rho$ var)
```

```
(value-of <<letrec double(x) = if zero?(x)
                               then 0
                               else -((double -(x,1)), -2)
            in (double 6)>> $\rho_0$)

= (value-of <<(double 6)>>
    (extend-env-rec double x <<if zero?(x) ...>> $\rho_0$))

= (apply-procedure
    (value-of <<double>> (extend-env-rec double x
                           <<if zero?(x) ...>> $\rho_0$))
    (value-of <<6>> (extend-env-rec double x
                      <<if zero?(x) ...>> $\rho_0$)))

= (apply-procedure
    (procedure x <<if zero?(x) ...>>
      (extend-env-rec double x <<if zero?(x) ...>> $\rho_0$))
    ⌈6⌉)

= (value-of
    <<if zero?(x) ...>>
   [x=⌈6⌉](extend-env-rec
                 double x <<if zero?(x) ...>> $\rho_0$))
```

Figure 3.10 A calculation with extend-env-rec

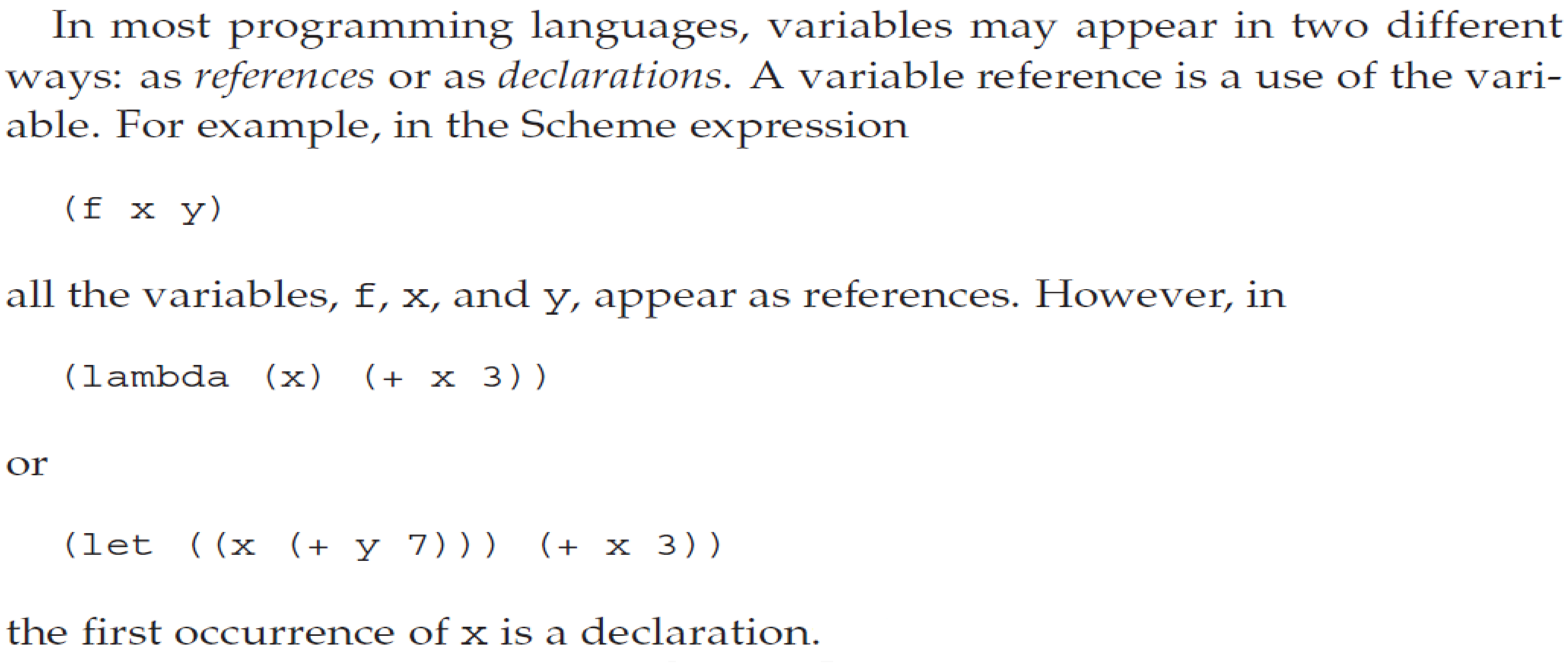
```
= (-
    (value-of
      <<(double -(x,1))>>
      [x=⌈6⌉](extend-env-rec
                           double x <<if zero?(x) ...>> ρ0))
    -2)

= (-
    (apply-procedure
      (value-of
        <<double>>
        [x=⌈6⌉](extend-env-rec
                           double x <<if zero?(x) ...>> ρ0))
      (value-of
        <<-(x,1)>>
        [x=⌈6⌉](extend-env-rec
                           double x <<if zero?(x) ...>> ρ0)))
    -2)

= (-
    (apply-procedure
      (procedure x <<if zero?(x) ...>>
        (extend-env-rec double x <<if zero?(x) ...>> ρ0))
      ⌈5⌉)
    -2)

= ...
```

Figure 3.10 A calculation with extend-env-rec

Figure 3.12 extend-env-rec added to environments.

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))

(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name b-var p-body saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var))))))
```

Scoping



In most programming languages, variables may appear in two different ways: as *references* or as *declarations*. A variable reference is a use of the variable. For example, in the Scheme expression

```
(f x y)
```

all the variables, `f`, `x`, and `y`, appear as references. However, in

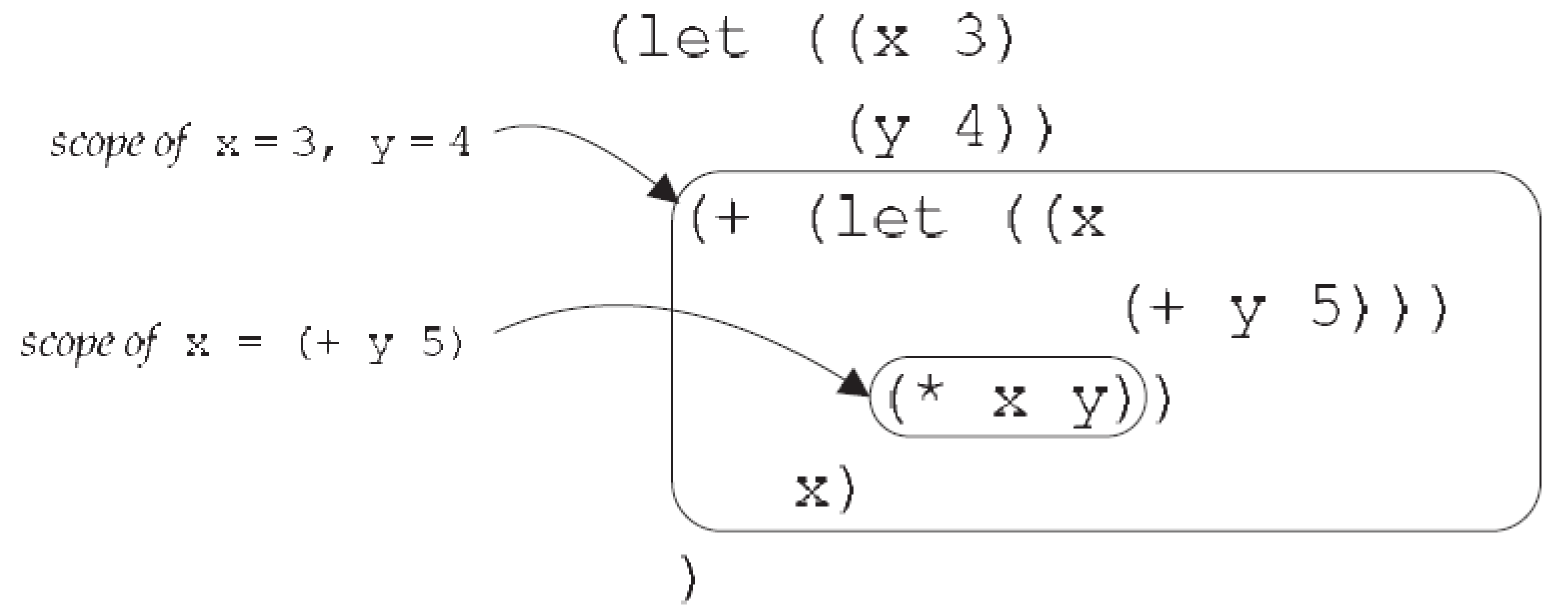
```
(lambda (x) (+ x 3))
```

or

```
(let ((x (+ y 7))) (+ x 3))
```

the first occurrence of `x` is a declaration.

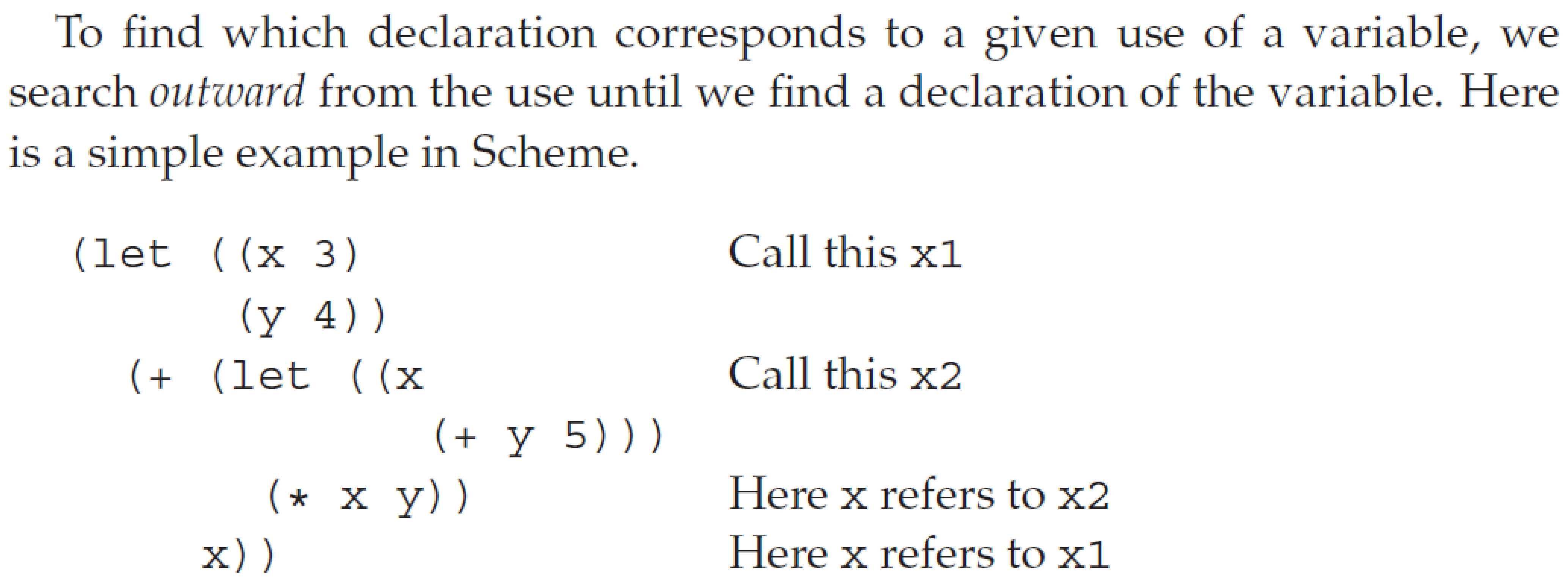
Scoping



```
(let ((x 3)
      (y 4))
  (+ (let ((x
            (+ y 5)))
       (* x y))
     x)
)
```

Figure 3.13 A simple contour diagram

…



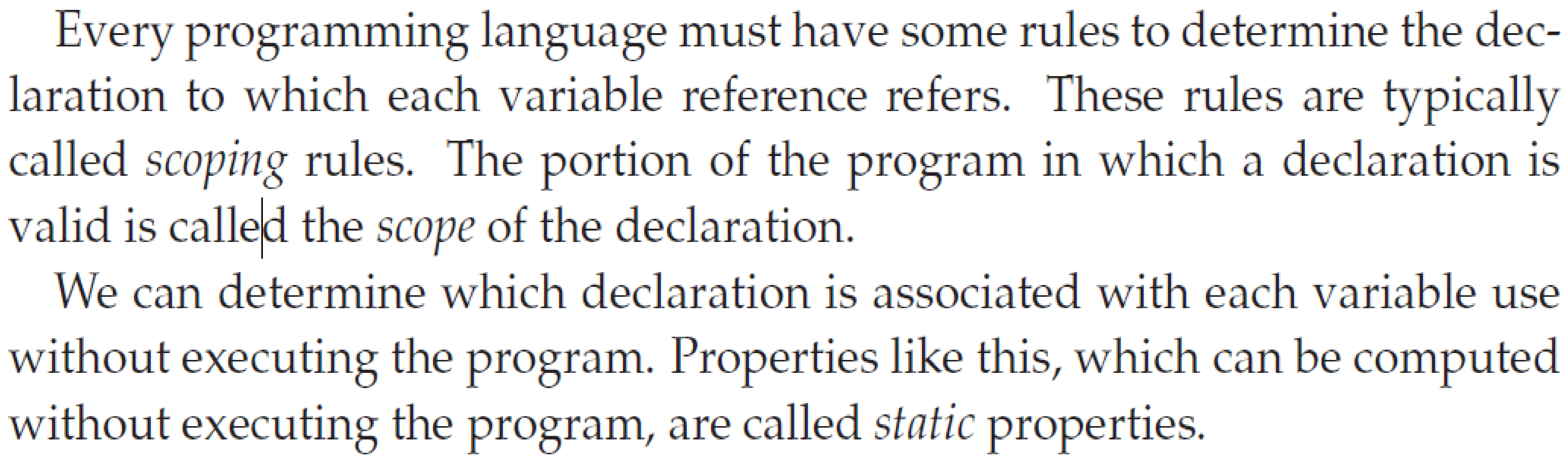
To find which declaration corresponds to a given use of a variable, we search *outward* from the use until we find a declaration of the variable. Here is a simple example in Scheme.

```
(let ((x 3)                          Call this x1
      (y 4))
  (+ (let ((x                        Call this x2
            (+ y 5)))
       (* x y))                      Here x refers to x2
     x))                             Here x refers to x1
```

In this example, the inner x is bound to 9, so the value of the expression is

```
(let ((x 3)
      (y 4))
  (+ (let ((x
            (+ y 5)))
       (* x y))
     x))

= (+ (let ((x
            (+ 4 5)))
       (* x 4))
    3)

= (+ (let ((x 9))
       (* x 4))
    3)

= (+ 36
    3)

= 39
```

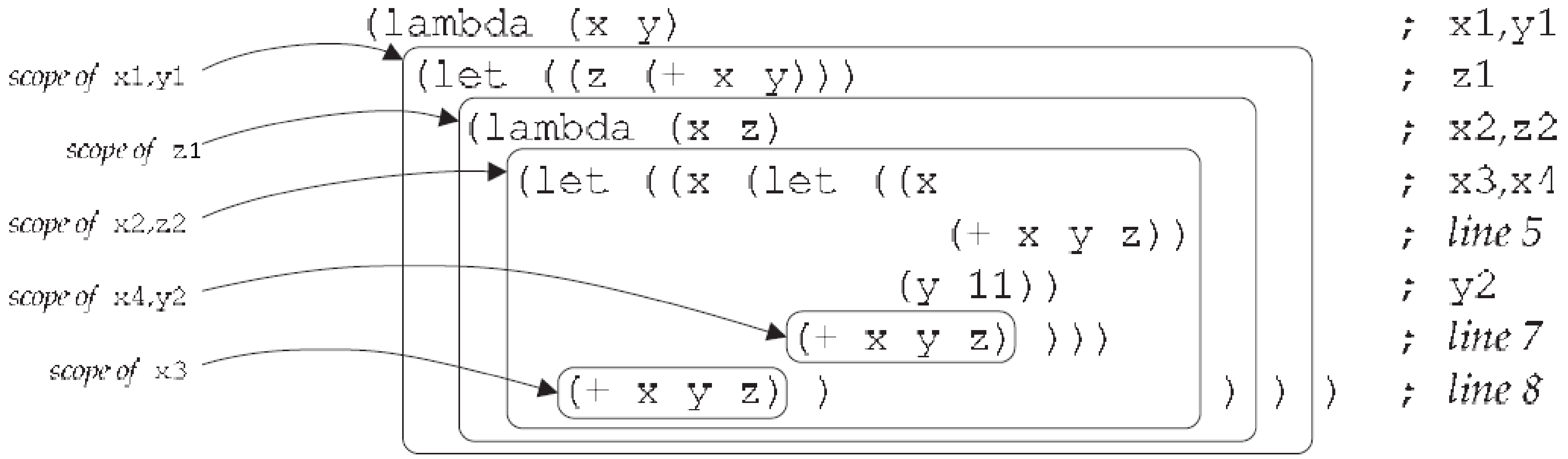
Static properties



Every programming language must have some rules to determine the declaration to which each variable reference refers. These rules are typically called *scoping* rules. The portion of the program in which a declaration is valid is called the *scope* of the declaration.

We can determine which declaration is associated with each variable use without executing the program. Properties like this, which can be computed without executing the program, are called *static* properties.

Contour diagram: Example



```
(lambda (x y)                                    ; x1,y1
 (let ((z (+ x y)))                              ; z1
  (lambda (x z)                                  ; x2,z2
   (let ((x (let ((x                             ; x3,x4
                   (+ x y z))                    ; line 5
                 (y 11))                         ; y2
              (+ x y z) )))                      ; line 7
    (+ x y z) )                   ) ) )          ; line 8
```

Figure 3.14 A more complicated contour diagram

…

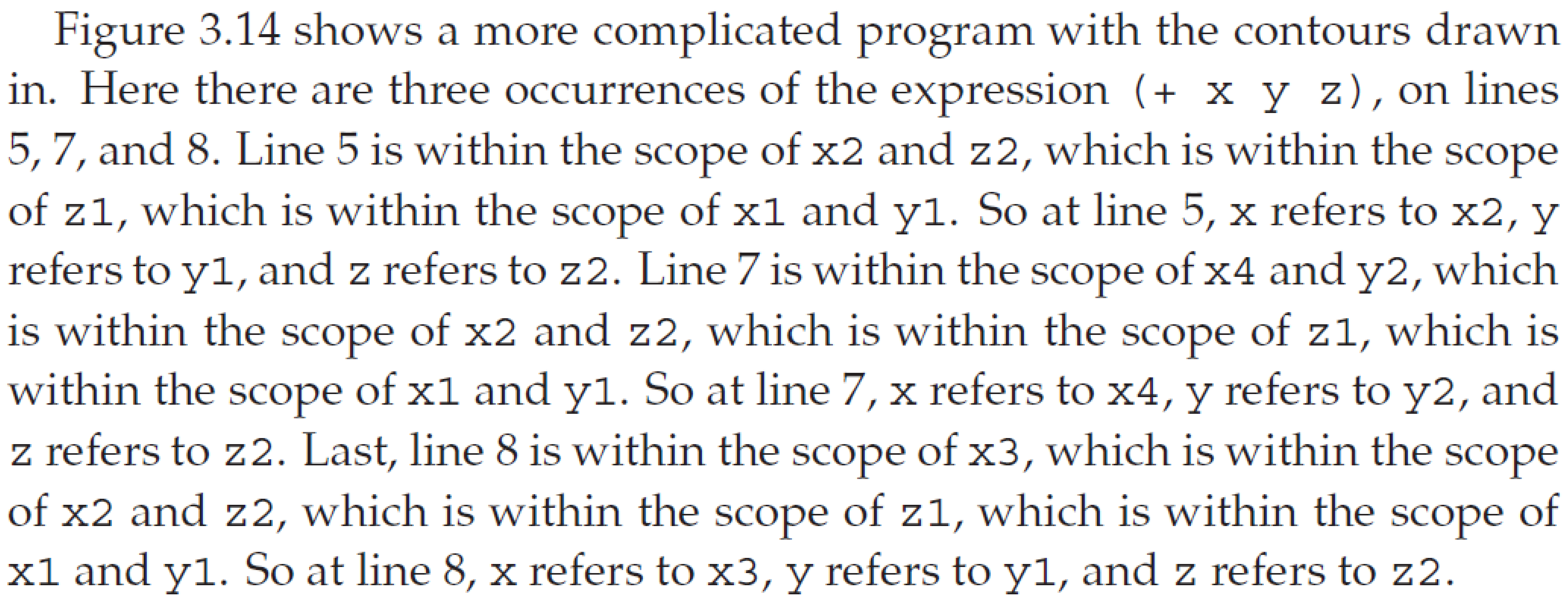


Figure 3.14 shows a more complicated program with the contours drawn in. Here there are three occurrences of the expression `(+ x y z)`, on lines 5, 7, and 8. Line 5 is within the scope of `x2` and `z2`, which is within the scope of `z1`, which is within the scope of `x1` and `y1`. So at line 5, `x` refers to `x2`, `y` refers to `y1`, and `z` refers to `z2`. Line 7 is within the scope of `x4` and `y2`, which is within the scope of `x2` and `z2`, which is within the scope of `z1`, which is within the scope of `x1` and `y1`. So at line 7, `x` refers to `x4`, `y` refers to `y2`, and `z` refers to `z2`. Last, line 8 is within the scope of `x3`, which is within the scope of `x2` and `z2`, which is within the scope of `z1`, which is within the scope of `x1` and `y1`. So at line 8, `x` refers to `x3`, `y` refers to `y1`, and `z` refers to `z2`.

Binding

The association between a variable and its value is called a *binding*. For our language, we can look at the specification to see how the binding is created.

A variable declared by a `proc` is bound when the procedure is applied.

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))
```

A `let`-variable is bound by the value of its right-hand side.

```
(value-of (let-exp var val body) ρ)
= (value-of body (extend-env var val ρ))
```

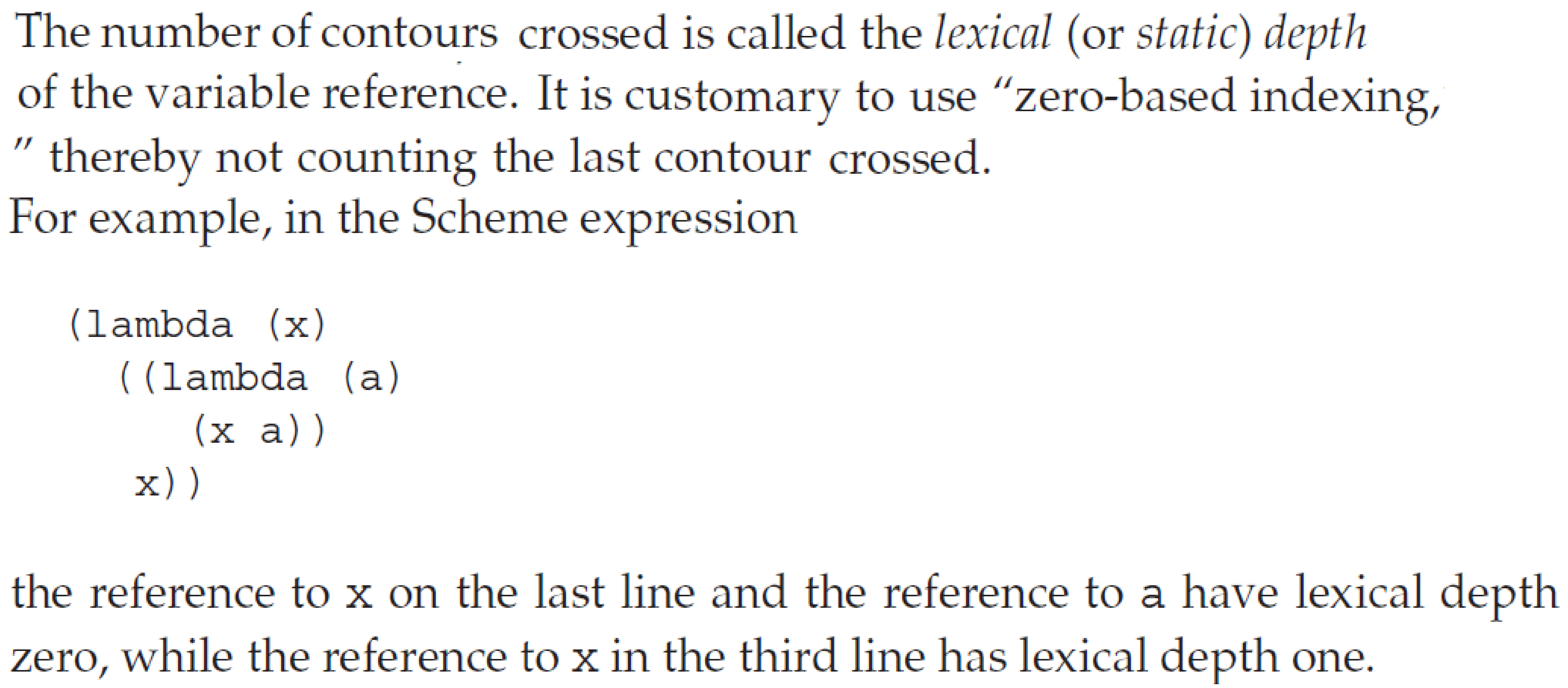
A variable declared by a `letrec` is bound using its right-hand side as well.

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

Extent of a binding

The *extent* of a binding is the time interval during which the binding is maintained. In our little language, as in Scheme, all bindings have *semi-infinite* extent, meaning that once a variable gets bound, that binding must be maintained indefinitely (at least potentially). This is because the binding might be hidden inside a closure that is returned. In languages with semi-infinite extent, the garbage collector collects bindings when they are no longer reachable. This is only determinable at run-time, so we say that this is a *dynamic* property.

Eliminating Variable Names



The number of contours crossed is called the *lexical* (or *static*) *depth* of the variable reference. It is customary to use “zero-based indexing,” thereby not counting the last contour crossed.
For example, in the Scheme expression

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

the reference to `x` on the last line and the reference to `a` have lexical depth zero, while the reference to `x` in the third line has lexical depth one.

Eliminating Variable Names

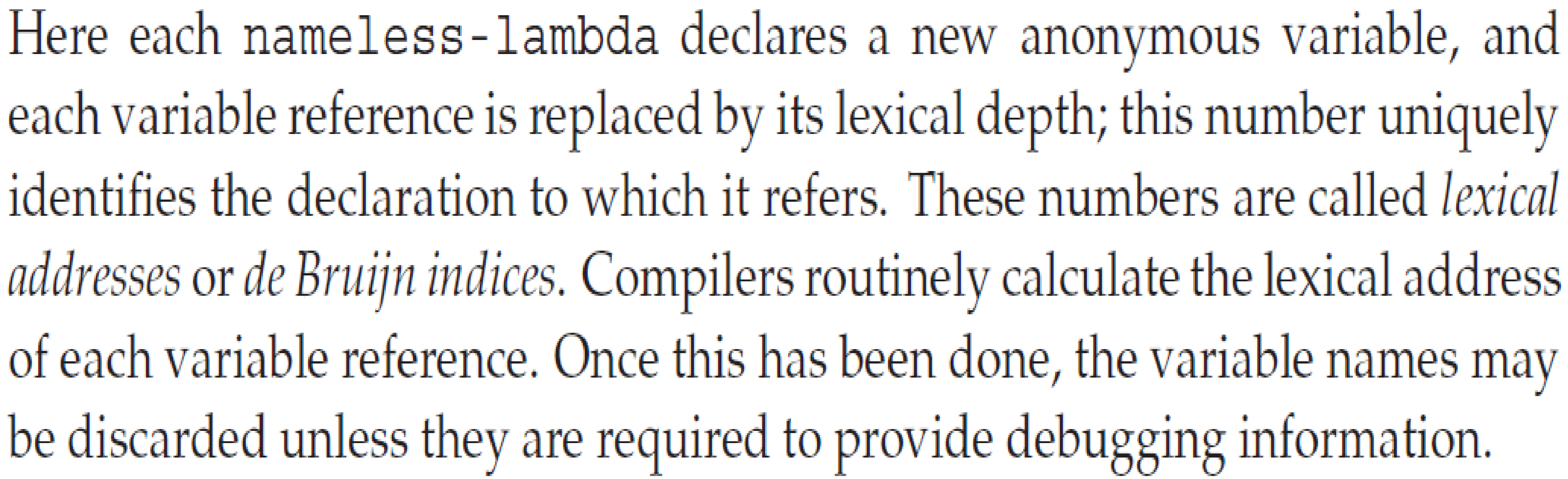
We could, therefore, get rid of variable names entirely, and write something like

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

$\longrightarrow$

```
(nameless-lambda
  ((nameless-lambda
     (#1 #0))
   #0))
```

Lexical Addresses



Here each `nameless-lambda` declares a new anonymous variable, and each variable reference is replaced by its lexical depth; this number uniquely identifies the declaration to which it refers. These numbers are called *lexical addresses* or *de Bruijn indices*. Compilers routinely calculate the lexical address of each variable reference. Once this has been done, the variable names may be discarded unless they are required to provide debugging information.

Lexical Addresses

Consider the expression

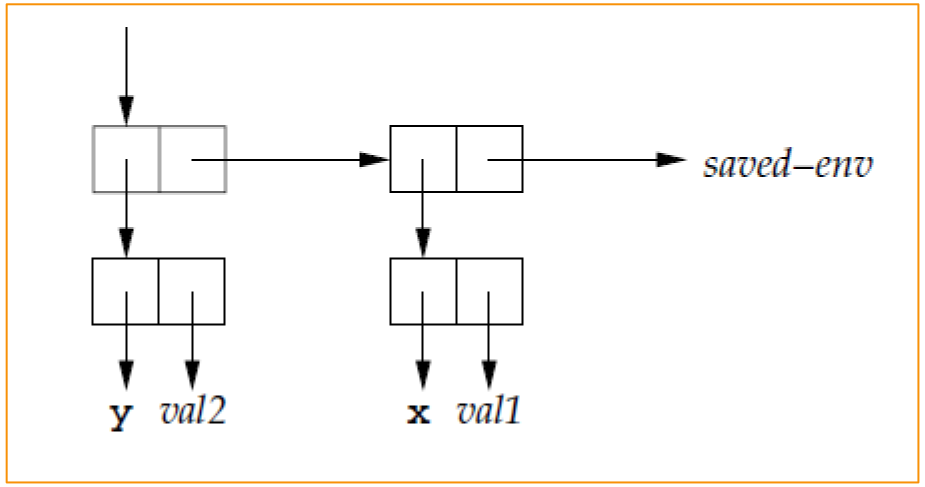
```
let x = $exp_1$
in let y = $exp_2$
   in -(x,y)
```

in our language. In the difference expression, the lexical depths of `y` and `x` are 0 and 1, respectively.

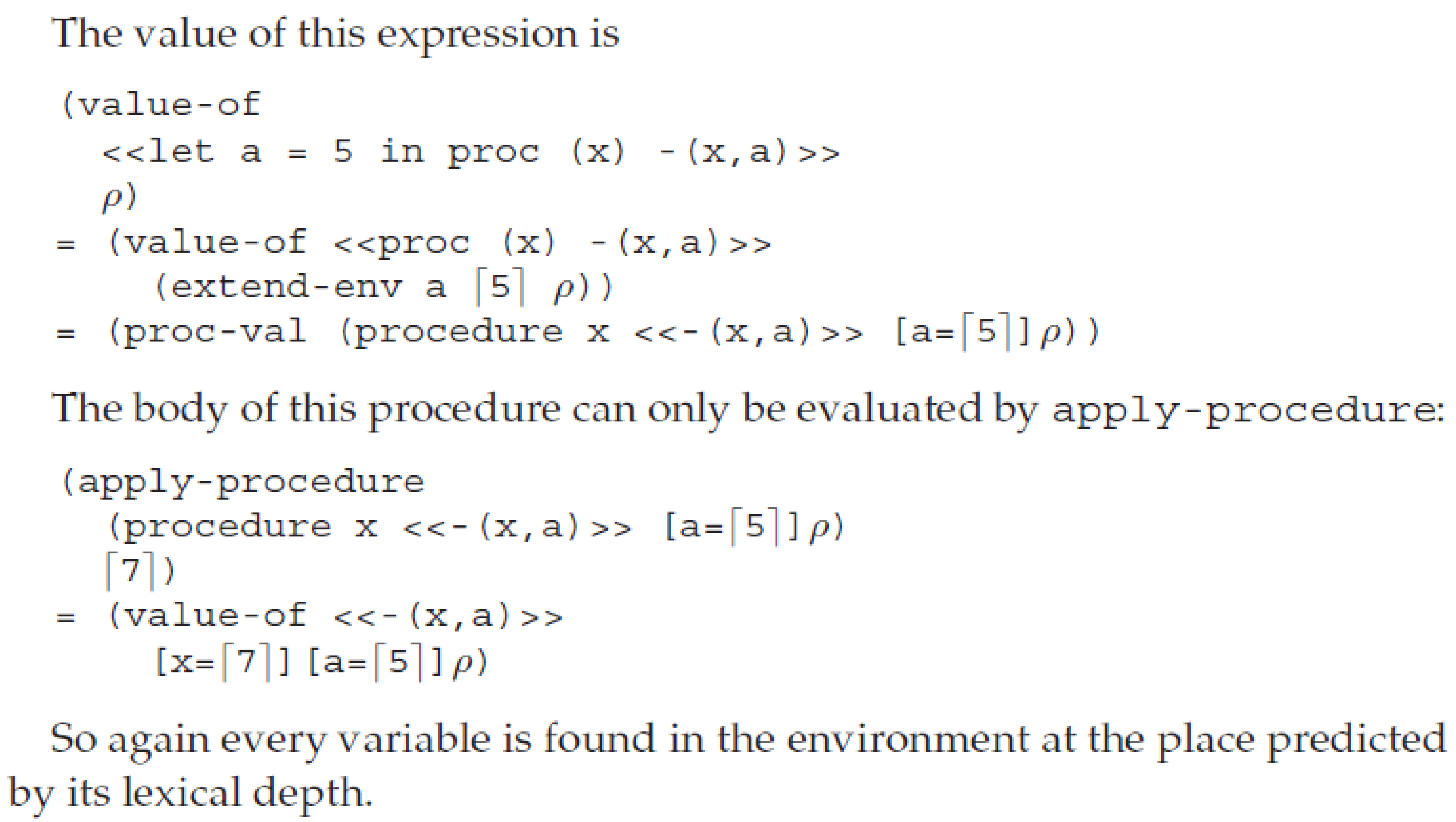
Now assume that the values of exp_1 and exp_2, in the appropriate environments, are val_1 and val_2. Then the value of this expression is

. . .

```
(value-of
  <<let x = exp₁
    in let y = exp₂
      in -(x,y)>>
  ρ)
=
(value-of
  <<let y = exp₂
    in -(x,y)>>
   [x=val₁] ρ)
=
(value-of
  <<-(x,y)>>
   [y=val₂] [x=val₁] ρ)
```



…



The value of this expression is

```
(value-of
  <<let a = 5 in proc (x) -(x,a)>>
  ρ)
= (value-of <<proc (x) -(x,a)>>
    (extend-env a ⌈5⌉ ρ))
= (proc-val (procedure x <<-(x,a)>> [a=⌈5⌉]ρ))
```

The body of this procedure can only be evaluated by `apply-procedure`:

```
(apply-procedure
  (procedure x <<-(x,a)>> [a=⌈5⌉]ρ)
  ⌈7⌉)
= (value-of <<-(x,a)>>
    [x=⌈7⌉][a=⌈5⌉]ρ)
```

So again every variable is found in the environment at the place predicted by its lexical depth.

Implementing Lexical Addressing

We now implement the lexical-address analysis we sketched above. We write a procedure `translation-of-program` that takes a program and removes all the variables from the declarations, and replaces every variable reference by its lexical depth.

We then write a new version of `value-of-program` that will find the value of such a nameless program, without putting variables in the environment.

...

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

```
#(struct:a-program
   #(struct:nameless-let-exp
      #(struct:const-exp 37)
      #(struct:nameless-proc-exp
         #(struct:nameless-let-exp
            #(struct:diff-exp
               #(struct:nameless-var-exp 0)
               #(struct:nameless-var-exp 1))
            #(struct:diff-exp
               #(struct:nameless-var-exp 2)
               #(struct:nameless-var-exp 1))))))
```

The Translator

We are writing a translator, so we need to know the source language and the target language. The target language will have things like `nameless-var-exp` and `nameless-let-exp` that were not in the source language, and it will lose the things in the source language that these constructs replace, like `var-exp` and `let-exp`.

Expression ::= `%lexref` *number*

$\boxed{\texttt{nameless-var-exp (num)}}$

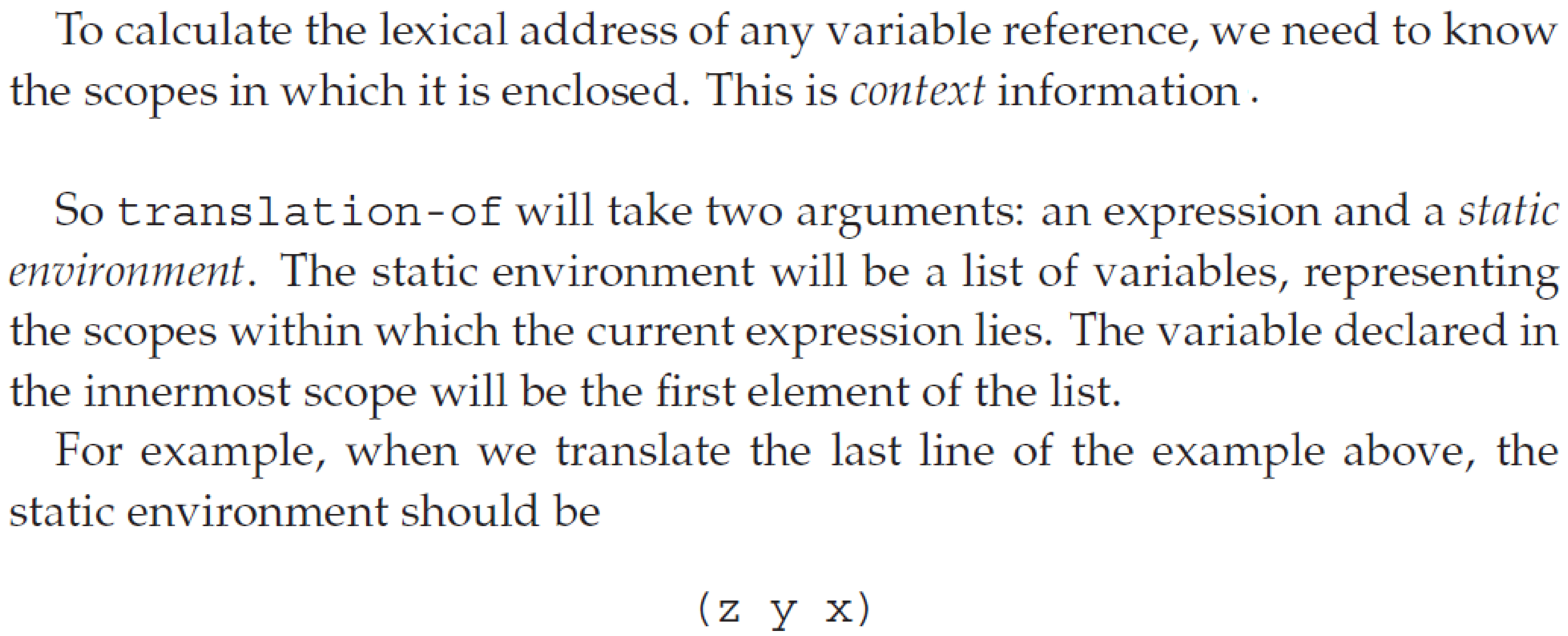
Expression ::= `%let` *Expression* `in` *Expression*

$\boxed{\texttt{nameless-let-exp (exp1 body)}}$

Expression ::= `%lexproc` *Expression*

$\boxed{\texttt{nameless-proc-exp (body)}}$

…



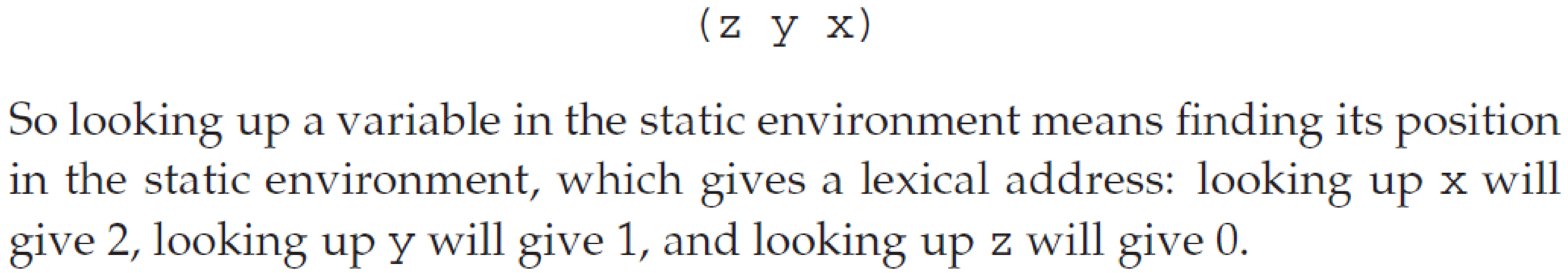
To calculate the lexical address of any variable reference, we need to know the scopes in which it is enclosed. This is *context* information.

So `translation-of` will take two arguments: an expression and a *static environment*. The static environment will be a list of variables, representing the scopes within which the current expression lies. The variable declared in the innermost scope will be the first element of the list.

For example, when we translate the last line of the example above, the static environment should be

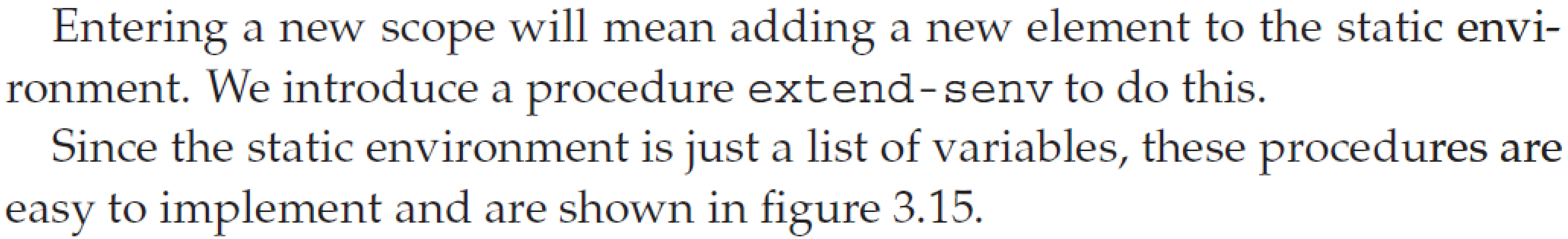
```
(z y x)
```

…



```
(z y x)
```

So looking up a variable in the static environment means finding its position in the static environment, which gives a lexical address: looking up `x` will give 2, looking up `y` will give 1, and looking up `z` will give 0.



Entering a new scope will mean adding a new element to the static environment. We introduce a procedure `extend-senv` to do this.

Since the static environment is just a list of variables, these procedures are easy to implement and are shown in figure 3.15.

Senv = *Listof*(*Sym*)
Lexaddr = *N*

empty-senv : $() \to$ *Senv*
```
(define empty-senv
  (lambda ()
    '()))
```

extend-senv : *Var* $\times$ *Senv* $\to$ *Senv*
```
(define extend-senv
  (lambda (var senv)
    (cons var senv)))
```

apply-senv : *Senv* $\times$ *Var* $\to$ *Lexaddr*
```
(define apply-senv
  (lambda (senv var)
    (cond
      ((null? senv)
       (report-unbound-var var))
      ((eqv? var (car senv))
       0)
      (else
        (+ 1 (apply-senv (cdr senv) var))))))
```

Figure 3.15 Implementation of static environments

Writing The Translator

1. Every `var-exp` is replaced by a `nameless-var-exp` with the right lexical address, which we compute by calling `apply-senv`.

2. Every `let-exp` is replaced by a `nameless-let-exp`. The right-hand side of the new expression will be the translation of the right-hand side of the old expression. This is in the same scope as the original, so we translate it in the same static environment `senv`. The body of the new expression will be the translation of the body of the old expression. But the body now lies in a new scope, with the additional bound variable *var*. So we translate the body in the static environment `(extend-senv` *var* `senv)`.

3. Every `proc-exp` is replaced by a `nameless-proc-exp`, with the body translated with respect to the new scope, represented by the static environment `(extend-senv` *var* `senv)`.

...

The procedure `translation-of-program` runs `translation-of` in a suitable initial static environment.

translation-of-program : *Program* → *Nameless-program*

```
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))
```

init-senv : () → *Senv*

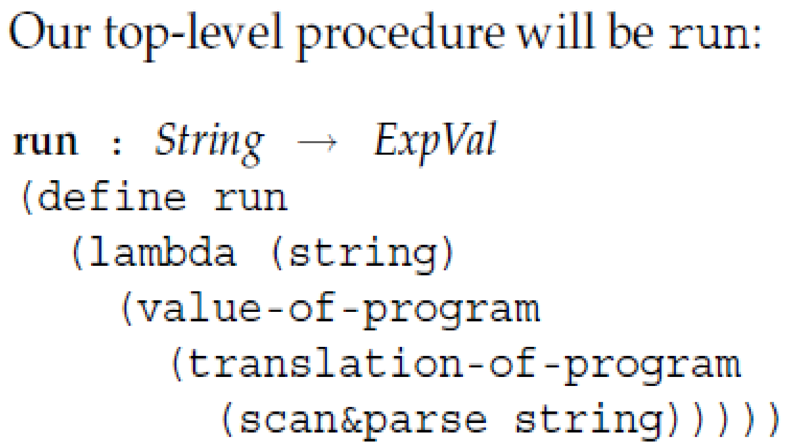
```
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

translation-of : *Exp* × *Senv* → *Nameless-exp*

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num) (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp)))))
```

Figure 3.16 The lexical-address translator

The Nameless Interpreter



Our top-level procedure will be `run`:

run : *String* → *ExpVal*

```
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string)))))
```

The Nameless Interpreter

Instead of having full-fledged environments, we will have nameless environments, with the following interface:

nameless-environment?	: $SchemeVal \to Bool$
empty-nameless-env	: $() \to Nameless\text{-}env$
extend-nameless-env	: $Expval \times Nameless\text{-}env \to Nameless\text{-}env$
apply-nameless-env	: $Nameless\text{-}env \times Lexaddr \to DenVal$

We can implement a nameless environment as a list of denoted values, so that `apply-nameless-env` is simply a call to `list-ref`. The implementation is shown in figure 3.17.

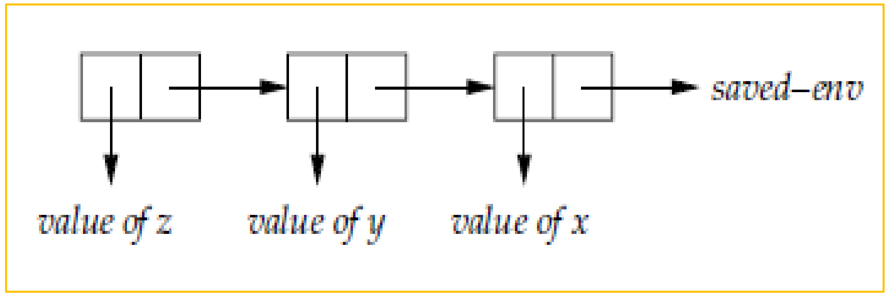


Figure 3.17 Nameless environments

nameless-environment? : *SchemeVal* → *Bool*
```
(define nameless-environment?
  (lambda (x)
    ((list-of expval?) x)))
```

empty-nameless-env : () → *Nameless-env*
```
(define empty-nameless-env
  (lambda ()
    '()))
```

extend-nameless-env : *ExpVal* × *Nameless-env* → *Nameless-env*
```
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))
```

apply-nameless-env : *Nameless-env* × *Lexaddr* → *ExpVal*
```
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

The revised specification for procedures is just the old one with the variable name removed.

```
(apply-procedure (procedure body ρ) val)
= (value-of body (extend-nameless-env val ρ))
```

We can implement this by defining

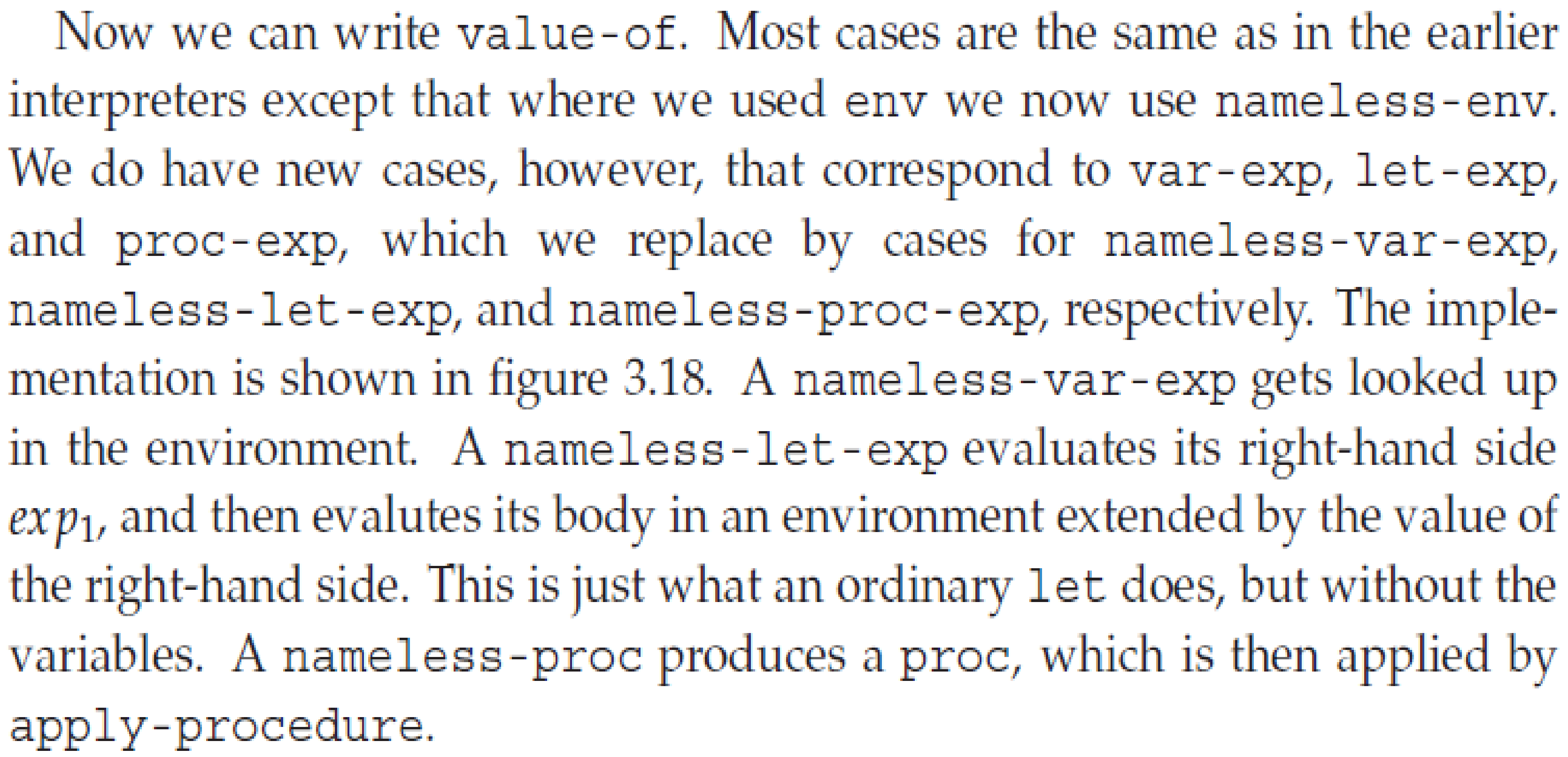
procedure : *Nameless-exp* $\times$ *Nameless-env* $\to$ *Proc*

```
(define-datatype proc proc?
  (procedure
    (body expression?)
    (saved-nameless-env nameless-environment?)))
```

apply-procedure : *Proc* $\times$ *ExpVal* $\to$ *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (body saved-nameless-env)
        (value-of body
          (extend-nameless-env val saved-nameless-env))))))
```

…



Now we can write `value-of`. Most cases are the same as in the earlier interpreters except that where we used `env` we now use `nameless-env`. We do have new cases, however, that correspond to `var-exp`, `let-exp`, and `proc-exp`, which we replace by cases for `nameless-var-exp`, `nameless-let-exp`, and `nameless-proc-exp`, respectively. The implementation is shown in figure 3.18. A `nameless-var-exp` gets looked up in the environment. A `nameless-let-exp` evaluates its right-hand side exp_1, and then evalutes its body in an environment extended by the value of the right-hand side. This is just what an ordinary `let` does, but without the variables. A `nameless-proc` produces a `proc`, which is then applied by `apply-procedure`.

Figure 3.18 `value-of` for the nameless interpreter

value-of : *Nameless-exp* × *Nameless-env* → *ExpVal*

```
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num)     ...as before...)
      (diff-exp (exp1 exp2)   ...as before...)
      (zero?-exp (exp1)     ...as before...)
      (if-exp (exp1 exp2 exp3) ...as before...)
      (call-exp (rator rand)   ...as before...)

      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp)))))
```